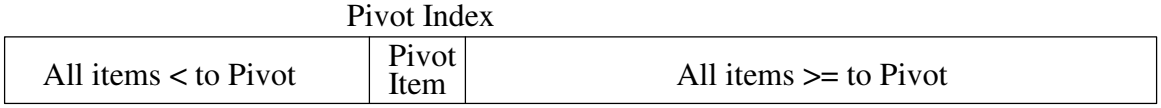
0) *Quick sort* is another advanced sort that often is quicker than merge sort (hence its name). The general idea is as follows. Assume "n" items to sort.

- Select a "random" item in the unsorted part as the *pivot*
- Rearrange (called *partitioning*) the unsorted items such that:



- Quick sort the unsorted part to the left of the pivot
- Quick sort the unsorted part to the right of the pivot

Given the following `partition` function which returns the index of the pivot after this rearrangement.

```
def partition(lyst, left, right):
    # Find the pivot and exchange it with the last item
    middle = (left + right) / 2
    pivot = lyst[middle]
    lyst[middle] = lyst[right]
    lyst[right] = pivot
    # Set boundary point to first position
    boundary = left
    # Move items less than pivot to the left
    for index in xrange(left, right):
        if lyst[index] < pivot:
            temp = lyst[index]
            lyst[index] = lyst[boundary]
            lyst[boundary] = temp
            boundary += 1
    # Exchange the pivot item and the boundary item
    temp = lyst[boundary]
    lyst[boundary] = lyst[right]
    lyst[right] = temp
    return boundary
```

a) For the list below, trace the first call to partition and determine the resulting list, and value returned.

	0	1	2	3	4	5	6	7	8	left	right	index	boundary
lyst:	54	26	93	17	77	31	44	55	20	0	8		

b) What initial arrangement of the list would cause partition to perform the most amount of work?

c) Let "n" be the number of items between left and right. What is the worst-case *O*( ) for partition?

d) What would be the overall, worst-case *O*( ) for Quick Sort?

e) Why does the partition code select the middle item of the list to be the pivot?

f) Ideally, the pivot item splits the list into two equal size problems. What would be the big-oh for Quick Sort in the best case?

g) What would be the big-oh for Quick Sort in the average case?

Consider the coin-change problem: Given a set of coin types and an amount of change to be returned, determine the **fewest number** of coins for this amount of change.

1) What "greedy" algorithm would you use to solve this problem with US coin types of {1, 5, 10, 25, 50} and a change amount of 29-cents?

2) Do you get the correct solution if you use this algorithm for coin types of {1, 5, 10, 12, 25, 50} and a change amount of 29-cents?

3) One way to solve this problem in general is to use a divide-and-conquer algorithm. Recall the idea of **Divide-and-Conquer** algorithms.
Solve a problem by:

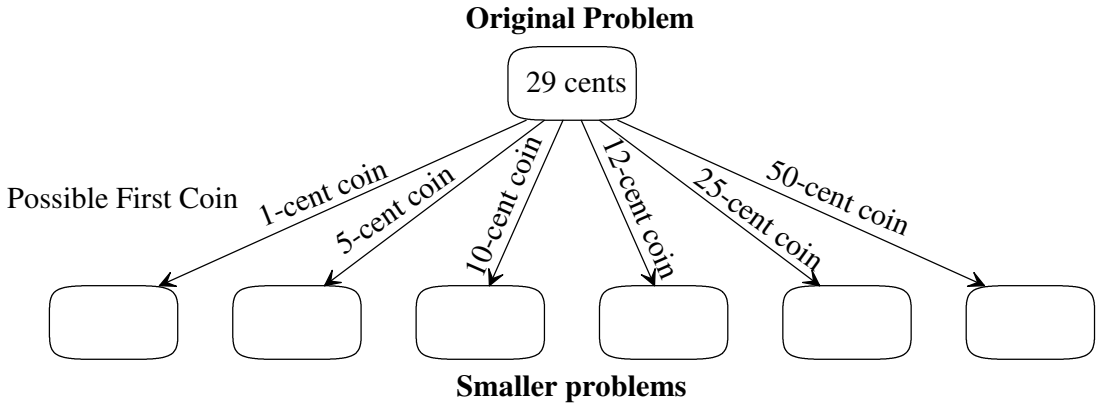
- dividing it into smaller problem(s) of the same kind
- solving the smaller problem(s) recursively
- use the solution(s) to the smaller problem(s) to solve the original problem

a) For the coin-change problem, what determines the size of the problem?

b) How could we divide the coin-change problem for 29-cents into smaller problems?

c) If we knew the solution to these smaller problems, how would be able to solve the original problem?

4) After we give back the first coin, which smaller amounts of change do we have?

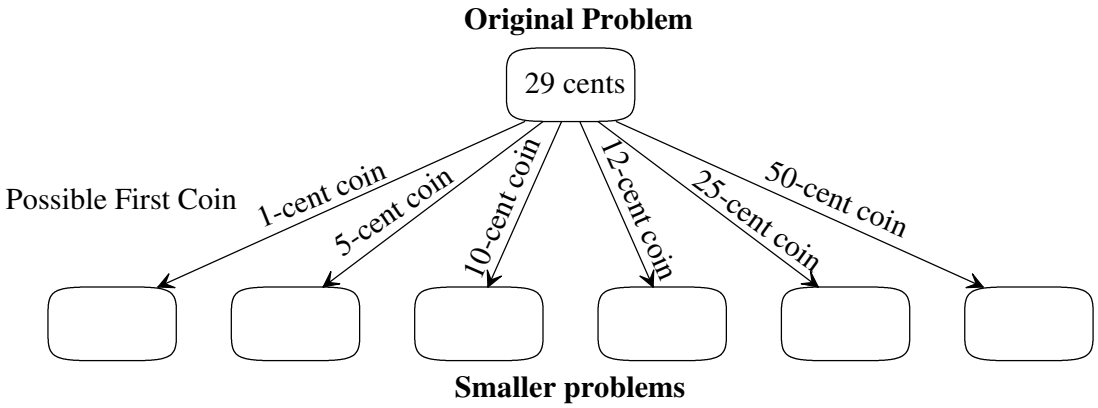


5) If we knew the fewest number of coins needed for each possible smaller problem, then how could determine the fewest number of coins needed for the original problem?

6) Complete a recursive relationship for the fewest number of coins.

$$\text{FewestCoins(change)} = \begin{cases} \min(\ \text{FewestCoins}(\qquad\qquad)\ ) + & \text{if change} \notin \text{CoinSet} \\ \quad \text{coin} \in \text{CoinSet} & \\ 1 & \text{if change} \in \text{CoinSet} \end{cases}$$

7) Complete a couple levels of the recursion tree for 29-cents change using the set of coin types {1, 5, 10, 12, 25, 50},



8) As in the maze problem of homework #5 we could use backtracking and a stack to maintain the list of unexplored choices, but we can also use recursion (with its run-time stack) to drive a backtracking algorithm. The general recursive backtracking algorithm for optimization problems (e.g., fewest number of coins) looks something like:

```
Backtrack( recursionTreeNode p ) {

    treeNode c;
    for each child c of p do                              # each c represents a possible choice
        if promising(c) then                              # c is "promising" if it could lead to a better solution
            if c is a solution that's better than best then   # check if this is the best solution found so far
                best = c                                  # remember the best solution
            else
                Backtrack(c)                              # follow a branch down the tree
            end if
        end if
    end for
} // end Backtrack
```

General Notes about Backtracking:

- the depth-first nature of backtracking only stores information about the current branch being explored so the memory usage is “low”
- Each node of the search space tree maintains the state of a partial solution. In general the state consists of potentially large arrays that change little between parent and child. To avoid having multiple copies of these arrays, a single “global” state is maintained which is updated before we go down to the child (via a recursive call) and undone when we backtrack to the parent.

For the coin-change problem:

a) What defines the current state of a search-space tree node?

b) When would a “child” search-space tree node NOT be promising?

Coin-change backtracking code in Python:

```
# global current state of the backtrack
numberOfEachCoinType = []
numberOfCoinsSoFar = 0
solutionFound = False
bestFewestCoins = -1
bestNumberOfEachCoinType = None

# profiling
backtrackingNodes = 0

def main():
    changeAmt = int(raw_input("Enter the change amount: "))
    coinTypes = raw_input("Enter the coin types separated by spaces: ")
    coinTypes = coinTypes.split(" ")
    for index in xrange(len(coinTypes)):
        coinTypes[index] = int(coinTypes[index])
    print "Change Amount:",changeAmt,"   Coin types:", coinTypes
    (fewestCoins, numberOfEachCoinType) = solveCoinChange(changeAmt, coinTypes)
    print "Fewest number of coins",fewestCoins
    print "The number of each type of coin in the solution is:"
    for index in xrange(len(coinTypes)):
        print "number of %s-cent coins is %s"%(str(coinTypes[index]),
                                               str(numberOfEachCoinType[index]))
    return

def solveCoinChange(changeAmt, coinTypes):

    def backtrack(changeAmt):
        global numberOfEachCoinType
        global numberOfCoinsSoFar
        global solutionFound
        global bestFewestCoins
        global bestNumberOfEachCoinType
        global backtrackingNodes
        backtrackingNodes += 1

        for index in range(len(coinTypes)):
            smallerChangeAmt = changeAmt - coinTypes[index]
            if promising(smallerChangeAmt, numberOfCoinsSoFar+1):
                if smallerChangeAmt == 0:  # a solution is found
                    # check if its best
                   if (not solutionFound) or numberOfCoinsSoFar + 1 < bestFewestCoins:
                        bestFewestCoins = numberOfCoinsSoFar+1
                        bestNumberOfEachCoinType = [] + numberOfEachCoinType
                        bestNumberOfEachCoinType[index] += 1
                        solutionFound = True
                else:
                    # update global "current state" for child before call
                    numberOfCoinsSoFar += 1
                    numberOfEachCoinType[index] += 1

                    backtrack(smallerChangeAmt)

                    # undo change to global "current state" after backtracking
                    numberOfCoinsSoFar -= 1
                    numberOfEachCoinType[index] -= 1
    # end def backtrack
```

```
    def promising(changeAmt, numberOfCoinsReturned):
        if changeAmt < 0:
            return False
        elif changeAmt == 0:
            return True
        else:  # changeAmt > 0
            if solutionFound and numberOfCoinsReturned+1 >= bestFewestCoins:
                return False
            else:
                return True

    # set-up initial "current state" information
    global numberOfEachCoinType
    global numberOfCoinsSoFar
    global solutionFound
    global bestFewestCoins
    global bestNumberOfEachCoinType

    numberOfEachCoinType = []
    for coin in coinTypes:
        numberOfEachCoinType.append(0)
    numberOfCoinsSoFar = 0
    solutionFound = False
    bestFewestCoins = -1
    bestNumberOfEachCoinType = None

    backtrack(changeAmt)
    return (bestFewestCoins, bestNumberOfEachCoinType)

main()
print "Number of Backtracking Nodes:", backtrackingNodes
```